Immer in Harmonie

Android-Laufzeitumgebung – Teil 2: Die Laufzeitbibliothek

Jörg Pleumann

Android-Applikationen werden überwiegend in Java implementiert. Die Laufzeitumgebung macht Android weitgehend zu einem gewöhnlichen Java-System. Die Einschränkung „weitgehend" signalisiert es schon: Es existieren offenbar Unterschiede zwischen einer Java-Umgebung von Oracle und der in Android. Für Entwickler sind diese durchaus interessant, insbesondere, wenn man Code zwischen beiden Plattformen portieren möchte.

Der erste Teil [Pleu10] dieser zweiteiligen Artikelserie zur Android-Laufzeitumgebung hat sich bereits der virtuellen Maschine (VM) gewidmet. Der vorliegende zweite Teil wirft einen Blick auf die Laufzeitbibliothek, die sogenannten *Core Java Libraries*. Bei diesen handelt es sich um Bibliotheken, die eng mit der VM verknüpft sind, also insbesondere Pakete aus den Namensräumen `java.*` und `javax.*`, die für das Funktionieren der Java-Programmiersprache erforderlich sind. Der Code ist in weiten Teilen eine Portierung und Anpassung des Open-Source-Projekts Apache Harmony [Harm].

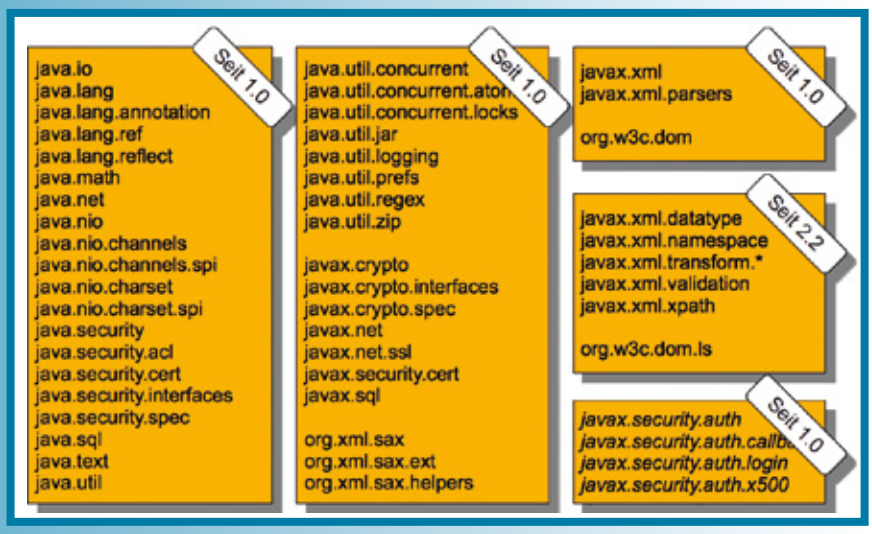

Abb. 1: Von Android unterstützte Java-Pakete

Abbildung 1 zeigt eine Liste der unterstützten Pakete, wobei alle bis auf die kursiv gesetzten komplett implementiert sind. Es ist offensichtlich, dass es sich hierbei (zum Glück!) nicht um ein abgespecktes Java im Sinne einer Micro Edition handelt. Die Auswahl an Paketen und Klassen orientiert sich vielmehr an einer Desktop-Implementierung von Java 1.5. So werden insbesondere die vier zentralen Pakete `lang`, `util`, `io` und `net` unterstützt. Auch viele andere nützliche Pakete wurden integriert, so etwa das Paket `java.sql` zur JDBC-Anbindung an die eingebaute SQLite-Datenbank oder das Paket `java.util.regex` für reguläre Ausdrücke. GUI-nahe Pakete sind nicht enthalten, da das Android-Framework diese Funktionalität bereitstellt.

Fast alle Pakete sind vollständig auf dem Stand eines JDK 1.5 und verhalten sich exakt so, wie von anderen Java-Implementierungen bekannt. Es gibt jedoch einige nennenswerte Unterschiede, die im Folgenden betrachtet werden.

Kernfunktionalität

Das zentrale Paket `java.lang` ist vollständig und kompatibel zu Implementierungen von Sun (jetzt Oracle) – anderenfalls dürfte es auch nur schwer möglich sein, Anwendungen für Android in Java zu entwickeln. Es gibt aber zwei nennenswerte Unterschiede:

- Da die Dalvik VM speziellen Bytecode erwartet, kann ein `ClassLoader` Bytecode nur aus `.dex`-Dateien laden. `.class`-Dateien werden nicht unterstützt. Zudem funktionieren die verschiedenen Varianten der Methode `defineClass()` nicht, da sie auf der Dalvik VM keinen Sinn ergeben. Einen teilweisen Ersatz liefert die Klasse `dalvik.system.PathClassLoader`.
- Die drei Methoden `suspend()`, `resume()` und `stop()` der Klasse `Thread`, die bereits seit Java 1.1 als `deprecated` markiert sind, werden nicht unterstützt. Leider werfen diese Methoden keine `Exception`. Sie sind vielmehr einfach leer implementiert, ohne dass dies (bisher) in der Android-Dokumentation vermerkt ist. Dies ist eine potenzielle Quelle für Probleme in Code, der Multithreading verwendet, speziell wenn dieser vom Desktop auf Android portiert wird.

Internationalisierung

Basis der Internationalisierung innerhalb der Core Java Libraries sind die *International Components for Unicode* (ICU), eine altgediente Open-Source-Bibliothek, die auch in anderen Systemen – z. B. praktisch allen Apple-Produkten bis hin zum iPhone – verwendet wird. Interessanterweise arbeitet auch das JDK selbst letztlich mit ICU. Dieser Code hat durch eine Schenkung von IBM bereits früh in der Geschichte von Java Eingang in die Implementierung von Sun gefunden, sich aber seitdem unabhängig von der Open-Source-Version von ICU entwickelt.

Android hingegen verwendet mit der Version 4.x praktisch den aktuellsten Stand von ICU und bindet diesen per JNI ein. Als Resultat können sich Klassen der Pakete `java.lang`, `java.util` und `java.text` trotz identischer Schnittstelle in Einzelfällen minimal anders verhalten als vom JDK gewohnt. Beim Parsen und Formatieren über `Formatter` und verwandte Klassen gibt es leichte Abweichungen in Syntax und Semantik. Ebenso kann eine `Locale` oder die Klasse `Character` andere Daten enthalten als gewohnt, wobei ICU generell eher um die Einhaltung aktueller Unicode-Standards bemüht ist als das auf Abwärtskompatibilität bedachte JDK.

Aus Platzgründen werden die meisten Gerätehersteller und Netzbetreiber zudem nur eine kleine Menge von Ländern und Sprachen in Android aufnehmen, sodass man sich nicht darauf verlassen sollte, dass jede exotische Region von jedem Gerät unterstützt wird.

Reguläre Ausdrücke

Auch das Paket `java.util.regex` ist von der Internationalisierung betroffen. Das mag im ersten Moment überraschen, hat aber einen simplen technischen Grund: ICU enthält ebenfalls eine sehr effiziente Implementierung von regulären Ausdrücken, sodass es sich anbot, die entsprechenden Java-Klassen auf diese abzubilden. Hier gilt ähnliches wie zuvor: ICU – und damit Android – ist syntaktisch und semantisch in allen wesentlichen Belangen kompatibel zum JDK. Die Bibliothek unterstützt einige Konstrukte, die das JDK nicht kennt.

Damit besteht eine – wenn auch geringe – Wahrscheinlichkeit, dass ein „normales" Zeichen in einem regulären Ausdruck als Metazeichen interpretiert wird. Das Problem lässt sich aber durch Escaping leicht lösen. Die Unterschiede zwischen den Implementierungen sind leider inzwischen nicht mehr in der Android-Dokumentation festgehalten. Im Zweifelsfall lohnt sich ein Blick auf die Dokumentation von ICU.

Sicherheit

Die Implementierung der Sicherheitspakete `java.security.*`, `javax.crypto.*`, `javax.net.ssl` und `javax.security.cert` ist in gewisser Weise hybrid. Primär wird intern die Open-Source-Bibliothek *BouncyCastle* als JCE-Provider verwendet. Damit stehen alle relevanten Algorithmen für Hashing (z. B. MD5) und Verschlüsselung (z. B. RSA) in einer reinen Java-Implementierung zur Verfügung. Zeitkritische Teile werden jedoch per JNI-Zugriff auf die Bibliothek *OpenSSL* „abgekürzt", die ebenfalls Teil des Software-Stacks von Android ist. Anders wäre eine effiziente Unterstützung von verschlüsselten Verbindungen und der Prüfung von Zertifikatsketten auf einem mobilen Gerät mit begrenzter Rechenleistung kaum möglich.

Für den Anwendungsprogrammierer ist dies an den meisten Stellen unsichtbar. Wer jedoch mit eigenen Keystores hantiert oder den Android-Keystore verändern möchte, sollte sich der Tatsache bewusst sein, dass die Laufzeitumgebung derzeit ausschließlich das Keystore-Format von BouncyCastle unterstützt. Da Android kein eigenes `keytool` besitzt, muss zur Konvertierung das JDK herangezogen werden.

Datenbank

Die Pakete `java.sql` und `javax.sql` sind vollständig implementiert, aber der in Android enthaltene JDBC-Treiber für die eingebaute SQLite-Datenbank nicht. Dies liegt teilweise an Beschränkungen der Datenbank selbst. So werden zum Beispiel die wesentlichen grundlegenden SQL-Datentypen auch von SQLite unterstützt, einige komplexere Typen – etwa Blobs – jedoch nicht. Derartige Zugriffe auf ein `ResultSet` haben eine `SQLException` mit einer entsprechenden Fehlermeldung zur Folge. Es ist aber möglich, das System auf dem üblichen Weg mit JDBC-Treibern für andere Datenbanken zu versorgen, die diesen Beschränkungen nicht unterliegen.

Es sei angemerkt, dass das Android-Framework mit `android.database.sqlite` ebenfalls über eine Schnittstelle zur SQLite-Datenbank verfügt. Diese ähnelt der JDBC-Schnittstelle, ist aber stärker auf das Programmiermodell von Android zugeschnitten. Insbesondere besitzt die Klasse `Activity`, welche die Basis für Applikationen bildet, eine Reihe von Hilfsmethoden zum komfortablen Zugriff auf Datenbanken, die mit einer Applikation verknüpft sind. Wer nicht auf Code angewiesen ist, der zwischen einem JDK und Android portabel ist, wird mit dieser Schnittstelle vermutlich besser bedient sein.

Extensible Markup Language

Die XML-Implementierung dürfte der Teil der Core Java Libraries sein, der den Beschränkungen mobiler Geräte am stärksten Rechnung trägt. Ursprünglich war sowohl die Menge der Pakete als auch der Umfang einzelner Klassen im Vergleich zu einem JDK 1.5 beschnitten:

- Bis einschließlich Android 2.1 waren nur die zentralen Pakete `javax.xml` und `javax.xml.parsers` vorhanden. Unterstützung für XPath oder Transformationen fehlte komplett. Das Document Object Model (DOM) war mit Level 2 Core auf dem Stand von etwa 2001. Version 2 des Simple API for XML (SAX) war ebenfalls enthalten, der XML-Parser von Android unterstützte aber nicht alle Merkmale davon. Die fehlende Validierung dürfte hier sicher der größte Wermutstrophen sein.
- Seit Android 2.2 sind auch die fehlenden Pakete um `javax.xml.transform` herum vorhanden. Das DOM wurde auf Level 3 Core gebracht, was die Unterschiede zum JDK nivelliert und eine Reihe von Convenience-Methoden beim Zugriff auf Knoten mit sich bringt. Die Parser-Implementierung ist gleich geblieben. Eine Implementierung für XPath und Transformationen existiert nach wie vor nicht, kann dem System jedoch über die übliche SPI-Schnittstelle hinzugefügt werden.

Zusammenfassung

Wie gezeigt, sind die Unterschiede zwischen der Laufzeitbibliothek von Android und der einer Standard-Java-Laufzeitbibliothek – insbesondere seit Android 2.2 – gering. Es ist leicht möglich, existierenden Java-Code auf Android zu portieren oder sogar gemeinsamen Code für beide Plattformen zu entwickeln, wenn man sich auf die unterstützten Pakete beschränkt. Zwar gibt es an einigen Stellen subtile Inkompatibilitäten. Diese sind aber gut zu umschiffen, wenn man sich ihrer bewusst ist.

Literatur

[Harm] Apache Harmony, `http://harmony.apache.org/download.cgi`

[Pleu11] J. Pleumann, Android-Laufzeitumgebung – Teil 1: Die (virtuelle) Maschine, in: JavaSPEKTRUM 5/2010, `http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2010/05/pleumann_JS_05_10.pdf`

Jörg Pleumann verfügt über langjährige Erfahrung im Bereich mobiles und eingebettetes Java. Er leitet die Android-Entwicklungsgruppe bei der Noser Engineering AG in Winterthur, Schweiz. Noser Engineering ist Gründungsmitglied der Open Handset Alliance, war unmittelbar an der Entwicklung von Android beteiligt und hat z. B. die Core Java Libraries und den größten Teil der offiziellen Compatibility Test Suite (CTS) beigesteuert.
E-Mail: joerg.pleumann@noser.com